Cooperative Component-Based Software Deployment in Wireless Ad Hoc Networks

Hervé Roussain, Frédéric Guidec

To cite this version:

Hervé Roussain, Frédéric Guidec. Cooperative Component-Based Software Deployment in Wireless Ad Hoc Networks. CD'05, Nov 2005, Grenoble, France. pp.1-16. hal-00341748

HAL Id: hal-00341748
https://hal.science/hal-00341748

Submitted on 25 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cooperative Component-Based Software Deployment in Wireless Ad Hoc Networks

Hervé Roussain and Frédéric Guidec

University of South Brittany, France
{Herve.Roussain|Frederic.Guidec}@univ-ubs.fr

Abstract. This paper presents a middleware platform we designed in order to allow the deployment of component-based software applications on mobile devices (such as laptops or personal digital assistants) capable of ad hoc communication. This platform makes it possible to disseminate components based on peer-to-peer interactions between neighboring devices, without relying on any kind of infrastructure network. It implements a cooperative deployment scheme. Each device runs a deployment manager, which maintains a local component repository, and which strives to fill this repository with software components it is missing in order to satisfy the deployment requests expressed by the user. To achieve this goal the deployment manager continuously interacts in the background with peer managers located on neighboring devices, providing its neighbors with copies of software components it owns locally, while obtaining itself from these neighbors copies of the components it is looking for.

1 Introduction

The number and variety of lightweight mobile devices capable of wireless communication is growing significantly. Such devices include laptops, tablet PCs, personal digital assistants (PDAs), many of which are now shipped with built-in IEEE 802.11 (a.k.a. Wi-Fi [1]) network interfaces. With such interfaces, the devices can occasionally be connected to an infrastructure network, using so-called access points that play the role of gateways. But the 802.11 standard also makes it possible for mobile devices to communicate directly in ad hoc mode, that is, without relying on any kind of infrastructure network. An ad hoc network is thus a network that can appear and evolve spontaneously as mobile devices themselves appear, move and disappear dynamically in and from the network [9].

For the users of laptops or PDAs, the prospect of deploying software applications on these devices as and when needed obviously appears as an attractive one, no matter if these devices communicate in infrastructure or in ad hoc mode. Yet, solutions for component-based software deployment have been proposed mostly for infrastructure-based environments so far, while very little effort has been devoted to software deployment in purely ad hoc networks.

In this paper we describe a model we devised in order to allow the deployment of component-based software applications on mobile devices participating in an ad hoc network. In Section 2 we motivate our approach by showing how infrastructure-based networks and ad hoc networks constitute radically different environments as far as software deployment is concerned, and we show that solutions that prove efficient

in infrastructure environments are hardly applicable in ad hoc environments. In Section 3 we present CODEWAN (COmponent DEployment in Wireless Ad hoc Networks), a middleware platform that implements our model. The main characteristics of this platform are discussed in Section 4, which also lists some directions we plan to work along in the future. In Section 5 we compare CODEWAN with other works that also address the problem of software deployment, either in infrastructure environments, or in ad hoc environments. Section 6 concludes the paper.

2 Motivations

In this section we show that deploying software components in an ad hoc network raises issues that usually do not appear in infrastructure networks. As a reminder, we first describe how software component provision and delivery are commonly performed in an infrastructure-based environment. We then show that an ad hoc network presents additional constraints that need to be addressed specifically.

2.1 Software deployment in an infrastructure network

Figure 1 illustrates a typical infrastructure network, including stable and mobile hosts—typically, workstations and laptops—interconnected through gateways (such as routers and switches). In such an environment some of the stable hosts can be in charge of storing components in so-called *component repositories*, and of implementing server programs capable of delivering these components on demand. Other hosts in the network can then behave as simple clients with respect to these servers. Whenever the owner—or the administrator—of one of the client hosts initiates the deployment of a new component-based software application on this device, the problem mostly comes down to locating the server—or servers—capable of providing the components required by this application, and downloading these components so they can be installed locally.

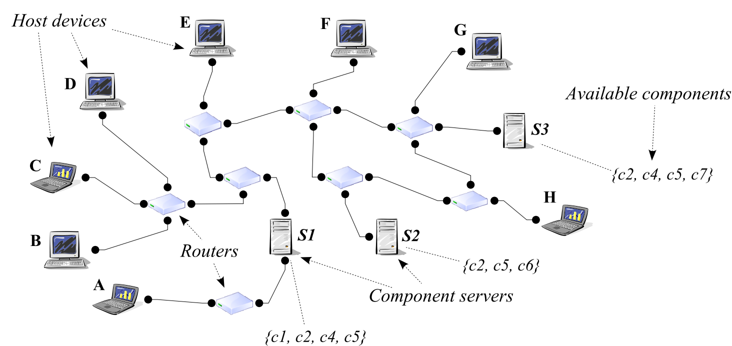

Fig. 1. Illustration of software component deployment in an infrastructure network

Consider the example shown in Figure 1, and assume that the owner of device *A* decides to initiate the installation on this device of an application that requires components *c1*, *c2* and *c3*. The deployment middleware running on device *A* must first identify

one or several servers capable of delivering these components. A component may actually be provided by several servers, for example in order to balance the workload in the network, or to allow fault tolerance. In any case, once a client has identified a server that can provide a component, obtaining this component simply requires its download between the server and the client. Note that in such a context the deployment of a component on a given host can usually be considered as a "real time" operation: once a user has ordered the deployment middleware to locate and download a component, this operation can usually be performed immediately. In the remaining of this section, we show that deploying components in an ad hoc environment can in contrast require a more lengthy process, which requires some middleware capable of enforcing a deployment strategy in the background on behalf of the user.

2.2 Software deployment in a dynamic ad hoc network

Figure 2 shows a typical dynamic ad hoc network, which consists of a collection of portable communicating devices. The devices in such a network are usually highly mobile and volatile. Device mobility results from the fact that each device is carried by a user, and users themselves move quite a lot. Device volatility is the consequence of the fact that, since the devices usually have a limited power-budget, they are frequently switched on and off by their owners.

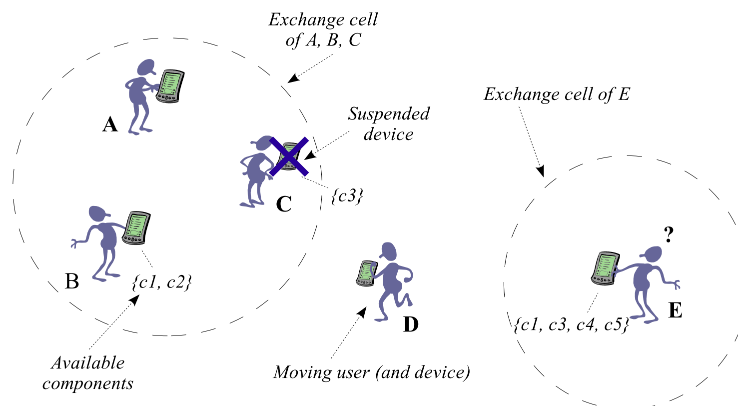

Fig. 2. Illustration of software component deployment in a dynamic ad hoc network

A major characteristic of wireless ad hoc networks is that communication interfaces have a limited transmission range. Consequently any device can only communicate directly with neighboring devices. Multi-hop transmissions can sometimes be obtained by implementing a dynamic routing algorithm on each device [10,12], but it is worth observing that even with dynamic routing, a realistic ad hoc network often presents itself as a fragmented network. Such a network appears as a—possibly changing—collection of so-called "islands" (also referred to as "clouds" or "cells" in the literature). Mobile devices that belong to the same island can communicate together, using either

multi-hop or single-hop transmissions depending on whether dynamic routing is used or not in the network. However, devices that belong to distinct islands cannot communicate together, because no transmission is possible between islands.

In such a context, a traditional client-server deployment scheme such as that illustrated in Section 2.1 is hardly applicable, as no device is stable and accessible enough to play the role of a server of components, maintaining a component repository and allowing client devices to access this repository whenever needed.

In the remainder of this paper, we present a model we propose in order to allow for these constraints. Basically, instead of being able to access a server whenever needed, each device must maintain a local component repository. A peer-to-peer interaction model then makes it possible for a device to cooperate with its neighborhood, by allowing its neighbors to obtain copies of the software components available on its local repository, while itself benefiting from a similar service offered by its neighbors.

Consider the example shown in Figure 2, and assume again that the owner of device *A* wishes to install on this device an application that requires components *c1*, *c2* and *c3*. In our example, *A* can obtain components *c1* and *c2* from device *B*. But as devices *C* and *E*—that both own a copy of component *c3*—are (possibly temporarily) unreachable, *A* cannot obtain a copy of component *c3* from any of these devices. Yet *A* could obtain component *c3* from device *C* if this device was switched on by its user. It could also obtain this component from device *E* if *A*'s user happened to walk towards *E*, or if *E*'s user happened to walk toward *A*. A roaming device such as *D* may even serve as a benevolent carrier between *E* and *A*, transporting component c3—and possibly other components as well—between separate islands, and thus contributing to the dissemination of software components and applications all over the network.

This example shows that when the owner of a mobile device participating in an ad hoc network requests the deployment of a component-based application on this device, there is no guarantee that this request can be satisfied immediately, as there is no guarantee that the components required for this deployment are readily accessible in the neighborhood. Yet, since the topology of an ad hoc network can change continuously and unpredictably as devices move and are switched on or off, the fact that a given component cannot be obtained at a given time does not involve that this component will remain inaccessible in the future. There is thus a need for some deployment middleware capable of ensuring the collection of missing components in the background in order to satisfy the user's needs.

3 Towards software component deployment on mobile devices

In this section, we present an overview of CODEWAN (*COmponent DEployment in Wireless Ad hoc Networks*), a platform we designed in order to support the deployment of component-based software applications on mobile devices communicating in ad hoc mode. CODEWAN implements a cooperative model, where neighboring devices interact in order to discover and exchange software components. Each device implements a local component repository, and a deployment manager is responsible for maintaining this repository on behalf of the user. Any component stored in the repository can be used to assemble and start an application locally. Copies of this component can also be sent on demand to neighboring devices.

3.1 Overview of the CODEWAN platform

The platform is built as a three-layer model, as shown in Figure 3. The upper and lower layers in this model have been described in details in [7] and [3] respectively. They are thus only described briefly below, and the paper then continues with a detailed description of the model's central layer, which implements the component repository and the deployment manager that maintains this repository.

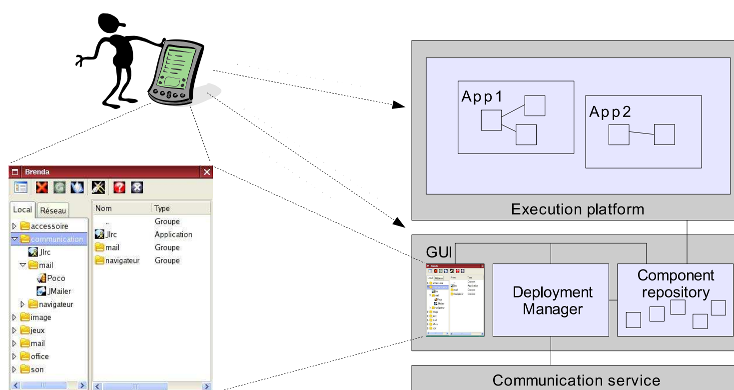

Fig. 3. Overview of the CODEWAN platform and screenshot of its GUI on a PDA

The upper layer in the platform is meant to provide a framework for assembling and running applications. Instead of defining its own component-model, CODEWAN was designed so as to rely on existing execution frameworks for component-oriented or service-oriented applications. In its current implementation it interfaces with JAMUS, a runtime framework that is primarily dedicated to hosting potentially malicious mobile applications [7], as well as with JULIA, an execution framework for applications designed using the Fractal component model [19].

The lower layer in our model was designed in order to support the asynchronous dissemination of so-called *transfer documents* in an ad hoc network. A transfer document is an XML document whose external element's attributes specify the conditions required for disseminating the document in the network. These attributes thus play approximately the same role as header fields in IP packets or in UDP datagrams. They indicate typically the document's source and destination, the expected propagation scope for this document, etc.

The "payload" of a transfer document consists of the internal XML elements that are embedded in the document. Any kind of structured information can be transported in a transfer document. In CODEWAN, though, transfer documents are used to transport software package descriptors in the network.

Figure 4 shows a typical transfer document. Attributes in this document indicate that it was sent by device *shiva*, and that it was addressed to any device in the neighborhood (notice that the communication layer CODEWAN relies on supports the use of wildcard addresses). The payload in this transfer document consists of a package descriptor, whose role and structure are detailed in Section 3.3.

```
<transfer-document
  document-id="fb54356fe468d9"
  source="device:shiva" destination="device:*"
  hops-to-live="3" lifetime="01:00:00"
  service-type="package-advertisement">
  <package-descriptor>
    <general-information
      type="application/java" category="communication/messaging"
      name="JMessager" version="1.3"
      provider="Laboratoire Valoria"
      summary="JMessager is a P2P messager"/>
    <java-application name="masc.jmessager.JMessagerImpl" />
    <dependencies>
      <required-package name="JMessengerUI" version="1.2"/>
      <required-package name="P2PAsyncDissemination"/>
      <optional-package name="AddressBook" version="2.0"/>
    </dependencies>
  </package-descriptor>
</transfer-document>
```

Fig. 4. Example of an XML transfer document carrying a software package descriptor.

The communication layer in CODEWAN provides services for encapsulating transfer documents in UDP datagrams. Large XML documents can be fragmented and then transported in distinct, smaller transfer documents that each can fit in a single UDP datagram. The communication layer of course supports the re-assembly of such fragments after they have been received from the network. Documents can be transferred either in unicast, broadcast, or multicast mode, and using either single-hop or multi-hop transmissions. In the latter case, all mobile devices in the network are expected to behave as routers, using algorithms for dynamic routing and flooding such as those described in [12,11,13].

Further details about CODEWAN's communication layer can be found in [3]. In the remainder of this paper we focus on the description of the central layer of the platform. The deployment manager is implemented in this layer, together with the component repository this manager is in charge of maintaining. The repository is a place where software components can be stored locally on a mobile device. Components stored in this repository are thus readily available for the execution framework that constitutes the upper layer of the platform. The deployment manager takes orders from the user, and interacts with peer managers that reside on neighboring devices in order to fill the local repository with components required by the user, while providing its peers with components they need in order to satisfy their own users.

3.2 Installation steps in CODEWAN

The deployment manager can provide the user with information about all the applications it knows about. At any time a given application is either:

- *installed locally* (meaning that this application is either already running in the local execution framework, or ready to be loaded and started in this framework);
- *installable* (meaning that all the components required for running this application are available in the local repository, so the application could be installed immediately if the user requested it);

– *not installable yet* (meaning that some of the components required by this application are not present in the local repository).

Besides observing the status of each application, the user can modify this status, requesting for example that an application be started (which implies that this application be already installed locally), or that an application be uninstalled (and all its components removed from the repository). Additionally the user can initiate the deployment of an application, thus instructing the deployment manager to try to obtain any missing component for this application from neighboring devices.

CODEWAN implements a basic user interface that can run in console mode. Additionally, graphical interfaces have been designed in order to facilitate the interaction between the user and the deployment manager running on a mobile device. For example Figure 3 shows an interface that was designed specifically for personal digital assistants.

3.3 Software components, applications, and packages

The deployment of component-based applications implies that components be transmitted in the network, and stored in local repositories. Before they can be loaded and executed in a runtime framework, software components are encapsulated in so-called *software packages*, that can be considered as storage and transfer envelopes for these components. Besides encapsulating the actual code of the components, software packages can additionally encapsulate some data required by a software component or application. They can also encapsulate documents describing the overall architecture of a component-based application (such as CCM component assemblies [8], or architecture descriptors in the Fractal model [19]).

Package descriptors Each software package in CODEWAN is associated with a package descriptor. This descriptor provides information about the package's identity, its content, its category, etc. It can be embedded in the package itself, but it can also be processed separately. For example, the transfer document shown in Figure 4 encapsulates a package descriptor as its payload. In a typical scenario such a document could be broadcast by a device in order to inform its neighbors about a software package that is available in its local repository.

The package descriptor shown in Figure 4 actually describes the main component of a Java messaging application (as specified by attribute *type* in the descriptor). It provides general information about the application, such as its name, version number, provider, etc. It also indicates that in order to be assembled the application requires components that can themselves be found in three other software packages. Two of these packages are absolutely needed for assembling the application, while the third one can be used optionally in order to improve the functionality of the application. This example shows that when the components encapsulated in a particular package depend on components that are encapsulated in other packages, this information is mentioned explicitly in package descriptors. Dependencies between packages can also appear when a package contains only the description of the architecture of an application, while other packages encapsulate components that are required for assembling this application, or data that are needed for running this application.

Software packages As mentioned above, software packages can encapsulate software components, as well as plain data or application architecture descriptions. A software package usually encapsulates its own descriptor, but this descriptor can also be extracted from the package and processed separately whenever needed.

In the current implementation of the CODEWAN platform, application data and the code of software components are encoded using the Base64 standard. The result of this encoding is then encapsulated as CDATA information in an XML document.

3.4 Main functionalities of the deployment manager

The deployment manager running on a mobile device is notably responsible for maintaining a local component repository on this device, while interacting with peer managers located on neighboring devices. Among other things the deployment manager can:

1. *decide what packages and package descriptors should be stored in the local repository and, if necessary, what packages and descriptors should be removed from this repository;*
   Notice that since mobile devices are usually resource-constrained, the deployment manager might sometimes have to reclaim the space occupied by unused, yet potentially interesting packages.
2. *announce to its neighbors what packages are available locally, thus indicating that these packages can be delivered on demand;*
   Announcing the availability of a package is performed by broadcasting a transfer document that encapsulates the descriptor of this package, as shown in Figure 4. Such an announcement can be broadcast periodically, or when a new device appears in the neighborhood. It can also be broadcast after a request has been received from a neighbor, as described in the next three items.
3. *search the neighborhood for specific packages, or for packages that satisfy precise criteria;*
   Package searching is performed by broadcasting a transfer document that encapsulates a “request for descriptors”. Such a request compares with a standard package descriptor, except that each attribute in the request defines a regular expression. Any deployment manager receiving a request can thus match this request against the descriptors of the packages stored on its local repository. If one or several of these descriptors match the request, then the deployment manager answers this request by announcing the availability of the matching packages, as described in item 2.
4. *discover what packages are available in the neighborhood;*
   This is performed by broadcasting a “request for descriptors” as explained in the former item, except that this request is not selective at all: it actually calls for the announcement of all the packages available in the neighborhood.
5. *ask a neighbor for the transmission of one—or several—particular package(s);*
   This is performed by sending the targeted neighbor a “request for packages”, which is similar to a “request for descriptors” except that the neighbor is expected to return the desired packages, rather than simply announce that it owns these packages. The actual transmission of software packages can be performed either in unicast, multicast, or broadcast mode, depending on the configuration of the sending deployment manager.

6. *receive packages from a neighbor, and decide for each package if it should be stored on the local repository.*
   The deployment manager can be expected to accept and store packages it has itself requested before. But since packages can sometimes be broadcast—as explained above—the deployment manager may also receive packages it has never requested. In such a case the deployment manager can be configured so as to implement a hoarding policy, storing packages that may prove interesting in the future.

The basic operations mentioned in the above list make it possible to devise and implement a number of different strategies for cooperative component deployment. Actually, while designing the CODEWAN platform we intentionally defined a comprehensive set of functionalities so as to allow a large number of interaction patterns between neighboring deployment managers. Several alternative deployment strategies can thus be implemented based on these functionalities. Part of our current work is now devoted to devising such strategies, and observing how they perform in realistic conditions.

Although a large number of deployment scenarios can be considered, the next section describes the major steps these scenarios can be based on.

3.5 Major steps in a deployment scenario

Learning about new applications At any time the deployment manager running on a mobile device maintains in the local repository a collection of application descriptors. As explained in Section 3.2 some of these descriptors correspond to applications that are not installable yet, meaning that some of the packages required for assembling these applications are not available locally. The deployment manager can thus "know" about the existence of an application (because it owns a descriptor of this application), even though this application is not yet installed locally.

A basic approach for a deployment manager to learn about new applications is simply to listen to the network in order to collect transfer documents that contain application descriptors, while broadcasting itself the descriptors of the applications stored in its repository. Neighboring deployment managers thus spontaneously inform each other about existing applications.

Initiation of a new application deployment In order to initiate the deployment of a new application on a mobile device, the user can rely on the interface of the deployment manager, and select with this interface an application that is not installed yet. This scenario however implies that the local deployment manager must already know about the existence of this application.

Alternatively a user may know about an application the deployment manager itself has never heard about. In such a case the user can inform the deployment manager about the name of this application, and the deployment manager will then start looking for the corresponding descriptor in the neighborhood.

Identification of missing packages Once the descriptor of the desired application is available, the deployment manager can examine the dependencies described in this descriptor in order to determine what other packages are needed for assembling this application.

Remember that several applications may be assembled out of the same set of components. The packages needed to assemble a new application may thus be already available locally, as they may have been collected before in order to assemble and start another application. Note also that the deployment manager may implement a hoarding policy, storing unused packages "just in case" in the local repository. Consequently, in the best circumstances, when determining what packages are needed for assembling an application the deployment manager may actually discover that all these packages are already present in the local repository. In such a case the deployment of the application can be considered as complete.

In most cases, though, when the user asks for the deployment of a new application the deployment manager is likely to discover that a number of required packages are missing in the local repository. For each application whose deployment is in progress the deployment manager maintains a list of desired packages (some kind of a "shopping list", actually). Once the packages required for a given application have been identified, their identity is appended to the corresponding "shopping list".

The deployment manager runs a background process that aims at collecting any package whose identity appears in at least one of the "shopping lists" it maintains.

Search for missing packages Searching for packages is a proactive operation that consists in broadcasting "requests for descriptors". This operation can be performed either periodically, or it can be triggered by an event, such as the detection of a new device in the neighborhood.

A request is a transfer document that contains a list of desired packages. Neighboring devices that own some of these packages are expected to reply by announcing the availability of these packages.

Note that since announcements are broadcast in the network, a deployment manager can sometimes discover passively that a number of packages it is looking for are available in the neighborhood. Packages can thus be located simply by listening to broadcast announcements. CODEWAN makes it possible to combine both forms of package discovery (proactive and reactive) in a single deployment strategy.

Download of missing packages Whenever the deployment manager discovers that some of the packages it is looking for are available on a neighbor device, it can react by sending a "request for packages", thus asking that the desired packages be transmitted in the network.

After receiving one of the packages it has requested, the deployment manager stores this package in the local repository, and removes its name from its "shopping lists". The descriptor of the package must also be analyzed in order to check if this package depends on other packages that are not available locally. If so, then these packages must also be considered as requested packages, and their names be appended to the deployment manager's "shopping lists".

Completion or termination of an application's deployment The deployment of an application is complete when the corresponding "shopping list" is empty, which means that all the packages required for assembling this application have been collected and

are now available in the local repository. The application can then be considered as installable, and be presented as such to the user through the user interface.

The user can also decide to cancel the deployment of a particular application at any time. In that case the “shopping list” maintained by the deployment manager for this application is discarded, and the packages that have already been collected and stored in the local repository are marked as unused (unless they are indeed used by another locally installed application, and unless their names appear in another local “shopping list”). Unused packages can be maintained by the deployment manager in the local repository as long as there remains enough space to receive and store other desired packages. Otherwise the deployment manager is entitled to remove unused packages whenever there is a need to free storage space in the repository.

4 Discussion and future work

4.1 Efficiency considerations

The model we propose for cooperative software deployment on mobile devices is inherently a probabilistic one. Indeed, when a user requests that a given application be deployed on a mobile device, there is no absolute guarantee that the deployment manager on this device will ever manage to collect the required packages. It is worth mentioning that this lack of guarantee is a consequence of the characteristics inherent to dynamic ad hoc networking, rather than a limitation of the model itself. However the model can be adapted in order to account for these constraints.

For example, in order to increase the chance that the requests of the user can be satisfied, the deployment manager in the CODEWAN platform was designed so as to exhibit a persistent behavior. Whenever it cannot obtain a number of packages from its current set of neighbors, the deployment manager simply persists and tries to obtain these packages later, after its neighborhood has changed. Device mobility and volatility thus become advantages in this process, as the neighborhood of a device is not limited to a fixed set of neighbors. Whenever a package cannot be found at a given time in the neighborhood, there is always a chance that it can be found in the future.

The actual efficiency of our model in realistic conditions depends on a large number of factors, such as the geographical distribution of mobile devices, their speed, the frequency at which devices are switched on and off by their users, data transmission rates, the amount of storage space available in each device’s local repository, the size of software packages, the number of packages required to assemble an application, etc. Work is now in progress in order to evaluate the average efficiency of our model in different conditions, based on simulations, and based on actual experimentation with CODEWAN-enabled mobile devices.

4.2 Towards adaptive software deployment

In the current implementation of the CODEWAN platform, the deployment manager running on a mobile device must be configured manually by the user of this device. For example it can be configured so as to announce periodically the packages it owns locally, and to broadcast periodically a request indicating the packages it is looking for.

In both cases, though, the user is responsible for choosing the appropriate periodicity for these transmissions.

The user must likewise determine how much storage space must be assigned to the local repository (which can be implemented either in memory or in the filesystem), and whether the deployment manager should implement a hoarding policy (storing in its local repository any package it receives from the network, even if this package is not mentioned in a local "shopping list").

Future work will notably focus on the development of a strategy manager capable of adjusting the behavior of a deployment manager transparently and continuously on behalf of the user. For example the periodicity for announcing local packages and requesting desired packages could be adjusted dynamically based on the mobility of a device, on observations of its neighborhood, or on internal events (such as the local device being suspended or resumed). The hoarding policy implemented by a deployment manager may likewise be guided by statistics about the requests received from the neighborhood: a deployment manager that frequently receives requests for a package it does not own locally may decide to try to collect this package so as to help multiply its copies—and thus its overall availability—in the ad hoc network.

4.3 Security considerations

The approach we propose for deploying software applications on mobile devices relies on the assumption that the owners of these devices may find it convenient to share software components with each other using ad hoc communication. This approach obviously raises a number of legitimate concerns regarding security, as the owner of a mobile device may for example be reluctant to run on this device pieces of software obtained from unidentified sources. We believe that this problem may be solved satisfactorily by using digital signatures so as to ascertain the origin of a software component, as well as ciphering in order to limit the use of a given component to a particular community of users. These are directions we plan to investigate in the near future.

4.4 Compatibility with standard component models and frameworks

CODEWAN is not strongly dependent on a specific execution framework, or on a particular component model. Actually the focus in this platform is put on the dissemination of software components rather than on the assembly and execution of component-based applications per se. In our opinion CODEWAN should quite easily accommodate almost any component model and any execution framework. The only condition is that components in the model considered can be transmitted and stored in packages, and that the execution framework can be adapted so as to take components from the local repository maintained by the platform's deployment manager, rather than from a legacy repository.

CODEWAN currently interfaces with two execution frameworks called JAMUS and JULIA. JAMUS is a security-oriented execution framework we designed, which provides a resource-constrained environment for untrusted Java applications [7]. JULIA is a framework that implements the Fractal component model [19]. Ongoing work aims at interfacing CODEWAN with OSCAR, a service-oriented framework for OSGi *bundles* [4].

5 Related work

Java Web Start [16] and Apache Maven [17] both support the deployment and the update of Java-based application programs. They are primarily meant to be used on stable, fully connected, infrastructure networks, though. They rely on a client-server model: a server (or a collection of servers in Maven) maintains a repository where applications can be stored, and clients can download new applications—or new versions of applications they have already downloaded—from this server.

A number of papers have proposed to apply the client-server model for software deployment in ad hoc networks. For example, JDRUMS [2] implements a content delivery system for software components. It relies on dedicated devices that host server programs called "JSTOREs". These server programs must register with a JINI lookup service in order to be located by the devices on which some software is to be deployed. Although mobile, pervasive devices are targeted in this work, the JSTOREs and the lookup service are assumed to be stable at any time, and available whenever needed.

As explained in Section 2 we believe that the traditional client-server model is hardly applicable for deploying and updating software in an autonomous ad hoc network, although it usually performs most satisfactorily in an infrastructure network. As an alternative to the client-server model we propose to rely on cooperative, peer-to-peer interactions between neighboring mobile devices. To our knowledge, this approach has not really been investigated so far, although cooperative software deployment has been considered in infrastructure-based environments, and proposals have been made to support code mobility or information dissemination in ad hoc networks.

SoftwareDock is a framework for distributed software deployment that uses mobile agents to support the transfer of software applications between so-called producers and consumers [5]. This approach thus compares with the client-server model. Moreover SoftwareDock is primarily meant to be used in infrastructure networks, as the prime motivation in this work is to allow load balancing and fault tolerance between software producers. Tacoma [15] is another system that relies on mobile agents to deploy components. Like SoftwareDock, though, it does not specifically address the problem of component deployment in ad hoc networks.

CORBA-$\mathcal{LC}$ defines the notion of CORBA Lightweight Component, and a number of design and implementation requirements for deploying such components are identified in [14]. This paper notably suggests that components should be deployed using a "peer network" model, where the whole network acts as a repository for managing and assigning resources (including components). However, although [14] observes that spurious node failures and node disconnections should be supported, our understanding of this paper is that it too considers the deployment of components in a quasi-stable, infrastructure-based environment.

Component deployment in ad hoc networks is specifically addressed in [6], which describes a framework for service-oriented computing. The components considered in this framework are actually proxy components, which must be deployed locally in order to allow local clients to access remote services. Service directories and implementation repositories are constructed and maintained using a distributed approach that implies the opportunistic collaboration of neighboring hosts in the ad hoc network.

SATIN provides support for component-based, self organized systems on mobile devices [18]. It supports the storage and the execution of components on a device, as

well as component advertisement, discovery and transfer between devices. SATIN is meant to serve as a generic platform that offers self organization through logical mobility and componentization. As such it does not readily compare with CODEWAN, which addresses specifically the problem of software deployment on mobile devices. Yet we believe that SATIN could serve as a framework for developing a deployment system similar to CODEWAN. This system would be dedicated to SATIN components, though (as SATIN defines its own component model), just like CORBA-$\mathcal{LC}$ only considers the deployment of CORBA components. In contrast CODEWAN is somehow more versatile. It processes software packages (that can encapsulate any kind of components) rather than the components themselves, and it delegates the problems of locally assembling and running components to an associate execution framework.

6 Conclusion

In this paper we presented the CODEWAN platform, which is dedicated to the deployment of component-based software applications on mobile devices participating in an ad hoc network. CODEWAN implements a peer-to-peer, cooperative model for software deployment. With this model, each mobile device maintains a local repository that can host a number of software components. The components stored in this repository are available for the execution framework that constitutes the upper layer of the platform. Neighboring devices can also exchange copies of the software components they own based on a peer-to-peer interaction scheme. A deployment manager is responsible for maintaining the local repository on a device, for interacting with peer deployment managers that run on neighbor devices, and for collecting software components in order to satisfy the requests of the owner of the local device.

The CODEWAN platform was implemented in Java and is now fully operational. It currently interfaces with the execution frameworks JAMUS and JULIA, and it thus supports the deployment and the execution of untrusted Java applications [7], as well as that of applications designed using the Fractal component model [19]. CODEWAN should also be able to support the deployment of OSGi bundles in the near future, using the service-oriented framework OSCAR [4].

Ongoing work implies using this platform in realistic conditions in order to assess its efficiency, and in order to compare the results obtained with alternative deployment scenarios. Future work should aim at augmenting the platform's functionality, for example by integrating support for digitally signed and encrypted software components.

Acknowledgements

This work is supported by the "Conseil Régional de Bretagne" under contract B/1042/2002/012/MASC.

References

1. Information Technology, Telecommunications and Information Exchange between Systems, Local and Metropolitan Area Networks, Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. ANSI/IEEE Std 802.11, 1999.

2. Jesper Andersson. A Deployment System for Pervasive Computing. In *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, pages 262–270, San Jose, October 2000.
3. Frédéric Guidec and Hervé Roussain. Asynchronous Document Dissemination in Dynamic Ad Hoc Networks. In *Second International Symposium on Parallel and Distributed Processing and Applications (ISPA'04)*, pages 44–48, Hong-Kong, China, December 2004.
4. Richard S. Hall and Humberto Cervantes. An OSGi Implementation and Experience Report. In *IEEE Consumer Communications and Networking Conference*, Las-Vegas, USA, January 2004.
5. Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *International Conference on Software Engineering*, pages 174–183, 1999.
6. Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman. A Component Deployment Mechanism Supporting Service Oriented Computing in Ad Hoc Networks. Technical Report WUCSE-04-02, Washington University, Department of Computer Science, St. Louis, Missouri, 2004.
7. Nicolas Le Sommer and Frédéric Guidec. JAMUS: Java Accommodation of Mobile Untrusted Software. In *4th Nord EurOpen/Usenix Conference (NordU 2002)*, Helsinki, Finland, February 2002. Best Paper.
8. OMG. Corba components, version 3.0, June 2002.
9. Charles Perkins. *Ad Hoc Networking*, pages 2–3. Addison-Wesley, 2001.
10. Pavel Poupyrev, Masakatsu Kosuga, and Peter Davis. Analysis of Wireless Message Broadcast in Large Ad Hoc Networks of PDAs. In *Proceedings of the Fourth IEEE conference on Mobile and Wireless Communications Networks*, pages 299–303, 2002.
11. Pavel Poupyrev, Masakatsu Kosuga, and Peter Davis. Analysis of Wireless Message Broadcast in Large Ad Hoc Networks of PDAs. In *Proceedings of the Fourth IEEE conference on Mobile and Wireless Communications Networks*, pages 299–303, 2002.
12. Elizabeth M. Royer and Chai-Keong Toh. A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. *IEEE Personal Communications Magazine*, pages 46–55, April 1999.
13. Yoav Sasson, David Cavin, and André Schiper. Probabilistic Broadcast for Flooding in Mobile Ad Hoc Networks. Technical Report IC/2002/54, Swiss Federal Institute of Technology (EPFL), 2002.
14. Diego Sevilla, José M. García, and Antonio Gómez. Design and Implementation Requirements for CORBA Lightweight Components. In *Proceedings of International Conference on Parallel Processing. Workshop on Metacomputing Systems and Applications.*, pages 213–218, sep 2001.
15. Nils P. Sudmann and Dag Johansen. Software Deployment Using Mobile Agents. In Judith Bishop, editor, *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD 2002)*, volume 2370 of *LNCS*, pages 97–107, Berlin, Germany, June 2002. Springer.
16. Sun Microsystems. Java Web Start 1.5.0 Documentation, 2004.
17. The Apache Software Foundation. Apache Maven. http://maven.apache.org/.
18. Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich. SATIN: A Component Model for Mobile Self Organisation. In *CoopIS/DOA/ODBASE (2)*, pages 1303–1321, 2004.
19. Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In *7th International Symposium on Component-Based Software Engineering*, pages 7–22. Springer-Verlag Heidelberg, 2004.